VŠB – Technical University of Ostrava
Faculty of Electrical Engineering and Computer Science
Department of Computer Science

Individual Professional Practice in the Company

Absolvování individuální odborné praxe

2019

Adam Kořínek

VŠB - Technical University of Ostrava
Faculty of Electrical Engineering and Computer Science
Department of Computer Science

Bachelor Thesis Assignment

Student:	**Adam Kořínek**
Study Programme:	B2647 Information and Communication Technology
Study Branch:	2612R025 Computer Science and Technology
Title:	Individual Professional Practice in the Company Absolvování individuální odborné praxe
The thesis language:	English

Description:

1. The student shall undergo individual practice with company Alza.cz a.s.
2. The final report structure shall be the following:

a/ Specification of the professional orientation of the firm where the student underwent his/her professional practice, specification of his/her occupational category.
b/ The list of tasks the student was assigned in the course of his/her professional practice including their time spans.
c/ Methods chosen when dealing with the given tasks.
d/ Theoretical and practical knowledge and skills acquired in the course of his/her professional practice.
e/ Knowledge and skills the student were missing in the course of his/her professional practice.
f/ Results achieved in the course of his/her professional practice and the general assessment of them.

References:

In accordance with the tutor guiding the student's professional practice.

Extent and terms of a thesis are specified in directions for its elaboration that are opened to the public on the web sites of the faculty.

Supervisor: **doc. Mgr. Jiří Dvorský, Ph.D.**

Consultant: Bc. Martin Říha

Date of issue: 01.09.2018
Date of submission: 30.04.2019

doc. Ing. Jan Platoš, Ph.D.
Head of Department

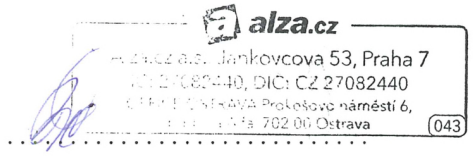
prof. Ing. Pavel Brandštetter, CSc.
Dean

I hereby declare that this bachelor's thesis was written by myself. I have quoted all the references I have drawn upon.

Ostrava, 30 April 2019 .....

I hereby agree to the publishing of the bachelor's thesis as per s. 26, ss. 9 of the Study and Examination Regulations for Bachelor's Degree Programmes at VŠB – Technical University of Ostrava.

Ostrava, 30 April 2019



I would like to thank everyone that helped me with this thesis. Without them, it would never exist.

Abstrakt

Tato práce popisuje průběh vykonávání bakalářské práce formou individuální odborné praxe ve firmě Alza.cz a.s. V průběhu praxe jsem plnil zadání a působil v roli vývojáře. Mým úkolem bylo přispět na tvorbě informačního systému sloužícímu pro administraci.

Klíčová slova: Bakalářská praxe, C#, SQL, Angular, Javascript

Abstract

This bachelor thesis describes the course of the individual practice at Alza.cz a.s. During my practice, I have fulfilled certain tasks and worked as a developer. The task was to participate in the development of the informational system used for the administration.

Key Words: Bachelor practice, C#, SQL, Angular, Javascript

Contents

List of symbols and abbreviations

CSS	–	Cascade Style Sheets
HTML	–	Hyper Text Markup Language
JS	–	Javascript
BL	–	Business Logic
IDE	–	Integrated Development Environment
API	–	Application Programming Interface
CLI	–	Command Language Interpreter
UI	–	User Interface

List of Figures

1 Practice environment

1.1 About Company

My practice was done in the Alza.cz a.s. Alza is an e-shop company operating in the Czech Republic and Slovakia and since 2014 It also operates in some other countries of the European Union. Alza focuses mainly on electronics, but also offers other goods than electronics like toys, drugstore, cosmetics, watches or sports gear.

The company was created in 1994 with the name of Alzasoft and was renamed to its current name in 2006. The owner and founder Aleš Zavoral still manages the company as the CEO. Currently, Alza.cz is the biggest and most profitable e-shop in the Czech Republic.

1.2 Admin team

Developers in Alza are divided into several teams like web development or administration. The biggest part of development teams is located in Prague and the Ostrava team focuses primarily on the administration.

In the Ostrava's admin development team are 6 front-end developers, 2 back-end developers, an architect, team-leader and 2 QA's. Usually, normal tasks are given to separate front-end developers since the backend part of the system is not too complicated and only in case of a bigger problem there is some kind of cooperation where back-end is all done by a back-end developer.

2 Used technologies

2.1 Microsoft SQL Server

Alza uses the Microsoft SQL Server is a relational database system developed by Microsoft. For stored procedures, we used the T-SQL language which is Microsoft's and Sybase's proprietary extension to the standard SQL language. It is used for interacting with relational databases.

2.2 .NET Core

As the server side language, Alza uses the .NET Core [1] framework. .NET Core is free, open-source managed software framework for Windows, Linux and, macOS. It was used primarily using the C# language that .NET Core fully supports.

2.3 Javascript (Angular)

As the client side language, we used with a few exceptions of JQuery, primarily the Angular [2] framework. It is originated from the AngularJS which was developed in 2012 and was build with the Model-view-controller concept. Later the whole language was redesigned with the name of Angular 2 in 2016. Since then, a lot of new versions were made and currently the newest version, the Angular 7 was used during my practice at Alza. It uses typescript instead of typical javascript, which allows the developer to use features like static typing, interfaces, classes, etc. It is dependant on the JS ES6. For handling of the project and its packages, usually NPM and Angular CLI is used.

2.4 Git

Git is the distributed version control system for tracking changes in source code during software development. It is used for the cooperation of developers on a single project, where each developer can work on a separate part of the project, then later merge them into the final project.

2.5 MVVM

In Alza the server side architectural pattern is the model-view-viewmodel(MVVM). It is used for separating the development of graphical user interface from the development of business logic. The MVVM was created by Microsoft in 2005 and is the variation of Martin Flower's presentation design pattern.

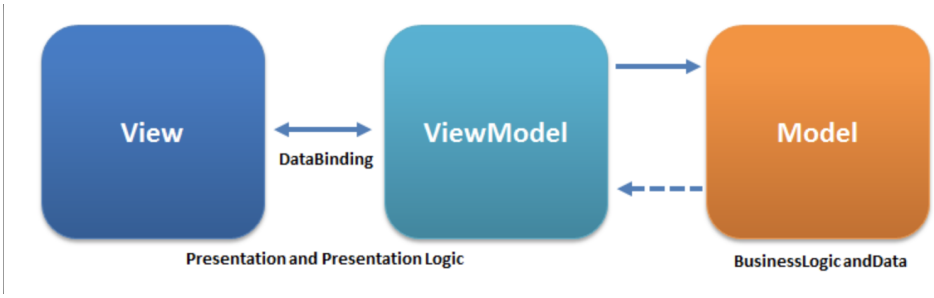


Figure 1: MVVM diagram
[3]

2.6 IDE's

As IDE's I mainly used the Visual Studio for the business logic and working with the .NET framework and I used its Git integration for handling the branches and git logic. For the SQL development, I used the Microsoft Management Studio and for the development of Angular, I used the Visual Studio Code.

3 First tasks

3.1 Styling bugs and fixes

First tasks were simple. It usually included fixing badly positioned labels, checkboxes, not responsive detail pages or collections or creating a small clipboard library. These tasks mostly required some knowledge of CSS and Bootstrap [4] library. If the problem was of a responsive kind, the solution required usage of the bootstrap grid system, or sometimes basic CSS styles mainly in the form of adding a margin or padding to a certain element.

3.2 Clipboard library

The biggest challenge of this part was probably to create a small library that was supposed to make things easier for people using the administration to for example copy certain codes to the clipboard so they can put them to for example Excel afterward. It was supposed to be a popup that would appear after hovering over the code. That would have an option to copy the code to the clipboard or add code to existing clipboard. First, we had to think through the design and how would It be positioned. To keep the design continuous It was heavily inspired by bootstrap default design. Because copying of a label is not possible in javascript, we had to create a temporary input field via javascript, take the text of the label we wanted to copy and insert it into the input field. Then call a function that would copy insides of input into the clipboard. Right after that, we encountered problems, because the browser will not allow the website to access users clipboard. So the only option was to override current clipboard every time a user wants to add something. For that, we created an array which we would rewrite if the user wants to copy only one text into the clipboard and add to it every time the user clicks the button for “adding to clipboard”. To make this feature really useful, we also want to show the user what items he has in his current clipboard. Because we cannot access users clipboard at his computer, we took data from an array he created by clicking the “copy to clipboard” button. UI consisted of a little bar on the right of a screen absolutely positioned, that on hover slides to the left and reveal users clipboard. This feature, of course, had to be global and easy to use for future usability on modules across the information system. We made this possible by making it as an angular component which only had one input, which was the text that was supposed to be copied.

4 Languages module

4.1 Requirements

The task was to rewrite module Languages from the old admin into the new one. Purpose of this module is to give users control over languages that Alza operates at. Users with the right administration rights would have access to a collection of languages as well as to the detail of each and every one of them. At the detail page, users would be able to alter the data to each country. In real life with this module, we say to which countries are we supposed to translate certain translation texts and which countries are or will be used at some point at Alza.

4.2 Thoughts and planning (collection)

When we first checked the database and data to this module, It was not too complicated. All the data came from one table and all we had to do was to follow development patterns and continuity in mind, create this module. The usual approach is to start with the collection and continue to detail page after the collection is finished.

4.3 Database procedures (collection)

First of all, I checked the database and the required fields, that are stored inside single table. To start with we need some data to work on, which required some SQL script to access the data and then get them via back-end business logic. Inside the procedure that would access the table, we would tell it to either create or alter the procedure. Then we would pass in the parameters that would handle filtering like specifying the number of items on one page, whether the items should be sorted in ascending or descending order and so on. Then we would do a simple select statement to select all the data from the table.

4.4 Model (collection)

After having the procedure completed and working, we have to access the data. This is done inside the Business logic layer of the information system via class so-called 'manager'. Due to rules of continuity, these class names start with the name of the module followed by whether it is for getting the collection of detail and finally by the manager. So in this case, we called the manager class LanguageCollectionManager. This LanguageCollectionManager inherits from the business collection manager, which the virtual void constructors for the basic selectors such as select, update, insert and delete, where we specify the data type to which we are going to map the data we are accessing through the procedure created previously. Inside this class, we created a method Select for selecting the collection of data. This function takes the parameters of the constructor LanguageCollection called businessCollection, Business connection and params that might we want to pass in. LanguageCollection constructor is the constructor where we call the

manager and then through which we can map it to the single element constructor. Business Connection is a generic class that is the part of business logic. In short, it handles the connection string to a database and also handles transactions and so on. Lastly, we have a set of parameters that we would like to enter. This is done by an array of objects. After that, we have to specify the SQL command. We can do this by creating variable 'command' equal to a SqlCommand constructor and pass in the name of the stored procedure and as a second parameter pass in the connection string. After this, we can finally access the database. We can do that by once again creating variable 'reader' equal to the variable of a SqlCommand constructor created before and then create while cycle inside which we can pass the condition to read while the reader.Read() is called. Condition breaks after there is no more data to access. This while cycle will loop through each row from a result query and access the data row by row. Inside this while cycle, we have to find out a way to assign the data that we received. The best available option is to have some kind of constructor class that would have the constructor properties inside of it. Which means we created the constructor Language class. Inside of it we created the datatypes and to each datatype, we created a constructor with the so-called auto property get and set. We can utilize the get property inside our select functions while loop. We will create variable language and make it equal to a new instance of the Language class and each property its value got from the DB. For example, assigning the ID would look something like language.Id = (int)reader['pkTblLanguage']". We can do this for each property and at the end, after all the properties were mapped, we can add this instance of Language to the businessCollection parameter. Once we have all the properties connected to the businessCollection, we can think about how to call all this action and actually display the data to the user.

4.5 Viewmodel (collection)

Calling the managers select method on a client-side would be sufficient to do inside some service where we would map data from the collection of our data. So as said, we created the service class called LanguageService inside which we created public PagedCollectionResult Get method. It would accept some sort of a filter model, by which we can specify sorting and search data. Because we do not need any complicated sorting we can just use default BaseCollectionFilter-Model which is the constructor for passing in the data like name or country and the default paging methods like specifying page and passing a sort key, page size or if we sort is ascending or descending. Inside the services Get method we want to initialize the pager. We can do that by creating variable pager and make it equal to CreatePager method called from the parameter. Also, we want to create a skip variable which would be equal to the pagers pageSize property and multiply it by pagers page property minus one. This serves as a skipper for the previous pages. For example, if we are on page three, we want to skip all the data that has been displayed on pages one and two and also it has to be dependant on the pageSize property as well so we do not skip more or less than required. Next step is to assign the collection we are getting via the LanguageCollection mentioned earlier. We can do this by assigning the LanguageCollections

instance to a variable and map it by data transfer object. After that, we have to return all this data. We can use PagedCollectionResult, which is a method that accepts the data as an IEnumerable and the total number of items that we have. To the IEnumberable we will pass the skipped items of the LanguageCollection, take the page size and convert all this data into an array. Then to the total number of items, we would pass the number of items we got from the collection and get the total number of items by calling Count function that C# already provides natively.

Lastly, we can call the services Get function and return the data in as a JSON result. This is done inside a newly created controller class. The controller class inherits from the base API controller, which is again a standard global class that initializes functions for converting a set of data to the JSON format. Also in this class, we should specify the URL that the requests are going to be sent to and will handle them or redirect them to the service layer. Method for handling our get request will have to define it first. We can do this using the native AspNetCore MVC library and then initializing the type of request right above our method. The method will be public IActionResult accepting the filter options. Using the MVC native FromQuery initializer. With this done, we can use base APIs controller function convert the data into JSON and return them.

4.6 View (collection)

4.6.1 Initial setup

Next up comes the visual side that the user will actually see. The angular application is already set, so we only need to create the module we are working on. For this, we have to create a new angular module and inside of it initialize component that would handle the HTML and the typescript file that would handle the user inputs and contain the logic. The module could be created by the native ng generate command provided by the angular CLI. This command followed by the schematics(in our case its module) we want to create the name of the module would create the language.module.ts file that in Angular framework handles the imports, exports, declarations, providers and mainly routing for our small project. To initialize this module inside the application we have to add it to the global Angular routing. The global routing module should usually be inside the root module of our application. Inside of that we only have to add the object specifying the module we created, its path and the auth guard. The path would be the URL by which user can access the web page, then we specify the loadChildren attribute that would point to our module and load its children, lastly, we will add the AuthGuard that is an additional layer of protection that checks the login credentials. After all of this done we can move to create the collection component. This is done again by the ng generate command followed by keyword component and the name of the component. In our case, it is called ‘collection’. Now we have the component, Angular CLI already added the component to our module, but we have to create the routing for it manually. For that to happen the angular router has to be imported

and then declared inside of the imports as Router.forChild which will accept the routes as a parameter. Router parameter is created by the routes constant that would have its data type specified as Routes and will be equal to an array of objects (routes to components). This route should specify the path and the component it is linked to. In our case, the path would be empty, because the collection component will be the base page.

4.6.2 HTML

Next step is the creation of the HTML and overall view that the user would see and would interact with. The user interface and overall design follows the Inspinia [5] admin theme that is based on bootstrap. This means to achieve continuity across the application we have to follow these design patterns that are already defined and therefore we do not have to design anything from scratch. The user experience is also highly restricted by the rules of continuity, so it is not necessary to create any user experience patterns on our own. All of this work still has to be done manually, so to begin with we should create breadcrumb for the page. Styling for the breadcrumb is done by the shared component, so our only job is to specify items inside and titles. At this point It has to be noted that all of the texts across the whole application are created as translation variables. For the translation ngx-translate library is used. It required some JSON file where these variables would be stored and then we can access them via the 'mustache' syntax notation specifying the variable as a string and followed by translate keyword that tells the application that the variable connected by pipe is supposed to be translated. Translations are generally stored inside the i18n folder. There of course our application should have its JSON file stored as well. With this we add the inner text to the items of the breadcrumb. It should indicate where are we currently according to the menu navigation. And then pass which item in the breadcrumb is currently active (the page we are currently on). After the breadcrumb initialization, we can move to the content of the page. There is the basic structure that defines the styles used on the page with their classes that are usually styles defined by the Inspinia template. Inside that we can define table that would hold our data and display them as its content. First we have to create the table with HTML's paired tag element and then initialize its head and body. Inside of head there would be one row containing the information about data stored underneath it as a columns. Then we have to specify the body element where we will have rows with data from the database. At this time, we have no data to insert in here, but It can be done after the initial HTML environment is set up. After that we want to create the pager that would handle the option to choose the amount of items that would be displayed on the page and also will allow us to go through the pages as well as seeing on which page we are currently on. For this the already created component can be used. What it does in short is basically accessing the properties and displaying them on page underneath the table with unified styling. Finally all that is left is to create the page footer. This is also done by the shared page footer component, that only handles the styling. Page footer stores buttons that can interact with the form of a page. On a collection page it is usually button for creating new items. This

is exactly what we need inside our footer, so we create the pair hyperlink <a> tag, where we will specify its styling, insert appropriate text and an favicon icon for better interface purposes. Later we will add the click event inside of the component for handling the creation of new item.

4.6.3 Typescript

Now we can move to the components typescript file where the data can be accessed. Inside of this file, we already have the main template created by the angular CLI. First comes the pagers initialization for this we have to import the pager service which is already created shared service created to make the HTTP requests easier and adding the ability to simply specify filters. In our case, there are no filters needed, because this language module will never have much data to access because It handles the languages at which Alza operates. We can access the data by specifying the URL the request is supposed to go to and then pass in the default paging properties like if the data are supposed to be sorted ascendingly or descendently and the default sort key. With the data accessed, we can insert the data inside our table using the [ngFor] syntax, that acts as a for a loop. There has to be specified a constant each element of the loop should be assigned to and then specify the data that should be looped through. After that, we can access the data separately in each cell of the table. After that, we have to create the function that handles accessing the detail of each data row inside of the table. The parameter for this method would be number or null because we would pass null when creating a new item. Inside of the function an If statement has to be made to check whether the id parameter is null or not. If the statement is truthy (id is provided), we would tell the router to navigate to the URL required followed by the id and then optional query parameters. If the statement would be false, there should be the same router navigation only instead of an Id, we would navigate to /new. With this function created we can add the event pointing to this function to the HTML elements. The first obvious choice is to add this to each row, so the user can enter the detail page by clicking on each cell in the row. This is done by click event where we would pass the redirect function created earlier and pass in the id as a parameter. The id can be accessed in the same manner as the data displayed in each cell and that's by accessing the property of the looped element. Also, we want to add this functionality to the button in the footer we created before, so we can add there the same click event with the function only this time, the parameter we are passing in would be a null value. With this done, the collection seems to be okay, now we just have to check whether the number of items is correct with the database and then we can move to the creation of the detail page along with its the business logic.

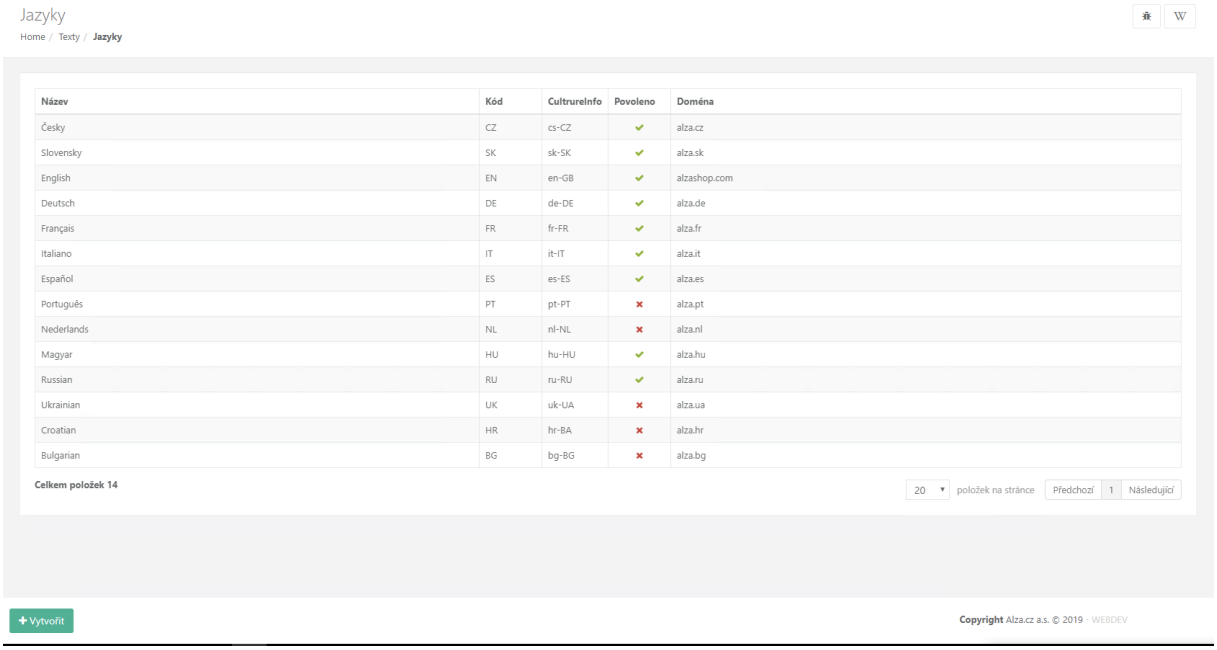


Figure 2: Language's collection view

4.7 Thoughts and planning (detail)

For the creation of the detail page, we, of course, have to get some kind of data that we could display to the user and further to let him alter the data according to the requirements. Once again the best approach would be to start with the SQL part and then continue passing the data to the view.

4.8 Database procedures (detail)

First, we would create a stored procedure that would handle getting the data. As done before we need to tell the procedure to either create itself if it is initialized for the first time or to alter itself if it already existed and we only need to change something inside of it. The difference here would be that we need to pass in some sort of parameter that would indicate what is the id of a certain item in our Languages collection that we want to get. For that to happen the only thing that needs to be done is to create simple select that would select required fields from the database and after that put a simple where condition to match the primary key of the table to the id of the item that we would like to access. Since we are already creating procedures for the detail, that is supposed to be interactive, the rest of crud(create, read, update, delete) should be created as well. We can start with the delete procedure since it would be the easiest one for the beginning. For deletion, the only thing we need to pass is the Id of an object we want to delete. For that the only thing we want to pass in is the id of an item that we would like to delete, we would do that similarly to the get procedure with only one id parameter. After that, the simple SQL delete statement would be enough with condition that tells that we want to delete the only

row with primary key matching our Id parameter. This all would, of course, be wrapped inside the begin and end syntax and would start with creating or altering the procedure and its name. Next up we have the update procedure. For the update procedure, we need to pass in all the data we want to update in our table, so all of that would be done through the parameters, which means we have to create one for each set of data we want to receive with matching datatypes. After that for assigning these data to the table, we need to create an update command where we will provide the name of the table we want to update and then the set operator where we would assign to each column the data the value of matching parameter. Then we only need to create a condition for matching the primary key with an id parameter so the procedure knows, which row we want to update. Lastly, we want to do the insert procedure. The Insert creation is a bit similar to the update. First we need to pass in all the data we want to insert as a parameter of the procedure, then an insert statement has to be created, inside of which we would specify the table we want to insert new record to and then specify columns and values that are supposed to be filled inside these columns as a new row. Usually you would have to take care of the primary key, but because our primary key column when the table was created, was given the increment property, we do not have to insert any kind of primary key, and we would not pass that in, because the increment would take care of that and would increment id by one from the last primary key that was inserted before. With all the procedures created, we can move to creation of the manager that would pass or get the data and trigger these procedures similarly to when we were creating it for getting the collection.

4.9 Model (detail)

For this, we need to create new manager either because it is better to distinguish manager that handles collections and a single set of data, but also because these are behaving differently and would unnecessarily overcomplicate the manager function making it more unreadable. With this said we can create the manager. It will inherit from the business persistent object that would have the Tobject set to Language constructor. Again the business persistent object is part of the business logic that is reusable and can be used as a helper to every manager and is here to handle the logic for the load or save methods called from the service. Firstly we will create the select object with protected accessibility and would be overridable. It will have receive parameters of the Language constructor, business connection and params object, similarly to the collections select method. After that, we will once again tell the SQL command to which procedure it should connect and fill up the parameters. In this select method case, we only need to pass in an id.

After that using reader, we will go through the data received, create a new instance of our Language constructor and again assign the data from the reader to the constructor with constructors get method. For the following three remaining CRUD methods, the flow will be the same only with differences in the name of the procedures and we will not need to use any reader. With delete, we will only specify the Id parameter and then call the execute query

command. Lastly, with the insert and update, things will be rather similar. We will only need to pass in the data we want to pass as parameters. The only difference there will be the name of the procedure we are calling and that we must not pass in the id with the insert, because that will be handled by the increment inside SQL. With the manager finished, we need to alter the constructor a little bit. We can still use the same one as we used with the collection, but this time we do not need to add it to any kind of collection, we only have one piece of data. For that we only need a simple load method, that would accept the id parameter and would create a new instance of itself with the Id passed in. This allows us to load a certain set of data by calling the load method with id as a parameter.

4.10 Viewmodel (detail)

That load method will be afterward called from the service layer of our application. There we will create another get method. It would accept the id parameter and would have a type of the data transfer object we created earlier. There we need to create a variable that would contain the single data for our detail. There are two options for that, the first one would be to call the collection and load all of the data, then call the first or default method built in by C# natively and find the item we are looking for by matching ids, or we can call the load method we created inside our constructor a few moments ago. For better performance we do not need to call more items than needed, so we would use the load method inside the language constructor and return it. After then we also need to create methods that would handle the put, post and delete calls. These are again similar because all of these are only supposed to send something, not receive. In order to do that we will create new public voids that would have the name of the update, create and delete. The create and update methods would accept the parameter of our language data transfer service object that will carry the data and then map the data transfer service to our constructor, then with the update method we would load the set of data as we have done in the get method and map its properties to our new set of data from the data transfer object. For the create method, we would not load any new data, instead, we would simply create a new instance of our language constructor and map the data from data transfer object onto it. With delete, we only need to accept the id, not the whole object, then load the language with our load method and call the delete method from the business logic onto it.

With that done, we have only one last layer to go before moving onto the angular part of our system. We need to create the controller that would once again get the HTTP requests. We already are handling the get request for the collection. On this layer, the single get would almost the same as the collection one. We have to define that it is an HTTP get and insert, that it would accept the parameter from the HTTP request. Then under that create an IActionResult that would call the get function from the service layer and return its result as a JSON. This time it is going to be the same with the delete method, there we only need to change the type from get to the HTTP's delete and then we can continue similarly with inserting the id as a parameter and then load the delete method from service passing the id as its parameter. Next

up we have the patch method where we once again define that it handles an HTTP patch request and will have an id as its request parameter. Also, it will have the from body parameter that will be in a data transfer object we have created earlier form that we will pass as a parameter to the update method inside our service. Lastly, we have to create a post method. For that to happen it has to have the specification that it will handle HTTP post request. Then it will accept the same parameter as the update method and passing it as a parameter to the call of the services create method.

4.11 View (detail)

Finally, we can move to the last part which is the angular and the view that the user will actually see. Once again this will require us to create new component inside our language model, that will be called detail with the same command used earlier for creating the collection. When this is done, the component was automatically imported to our model, but we then have to add it to the routes. This means doing the same as we did with our collection component, but inside the path, we have to tell the router, that it will be accepting the id inside its URL. After that, we can head to the typescript file of the detail component and inside the 'ngOnInit' method, that is required by typescript and handles input after initializing of the component. There we have to create a way to handle routing. We are able to do this by subscribing on the routers params that we passed inside the collection component. This also automatically does the get request to the controllers API from the router. Inside the subscribe we will then create a method that will have a parameter result, to which the data got from the router will be assigned automatically. Then inside of this function, we will check whether the result has an id specified. If not, it will identify it as a new item and will insert it into the URL. Then we will create the constructors with null values for the new item. If the parameter contains some id would send an API request to the server with the request of the data. This is in Angular done by an httpClient native library for API requests. We are going to use this library to make the API request when passing in the id after the URL that we specified inside out controller. Then we can just subscribe on the request, pass in a parameter that would the result be assigned to. We can now create a new public global variable that we would assign the result of the request. Now when we have the data stored inside the variable, we can move to the HTML and actually display it to the user. For the handling of forms, we are using the Angular's native reactive forms library, that provides a model-driven approach to handling form inputs whose values will change over time. With this, we can create our form tag input, inside of which we will specify the name of the form. Then we will insert the breadcrumb heading as we have done before and add appropriate links and texts to it. After that, we can move on to creating the input fields that will display our data and style them. We will use the basic bootstrap styling for those input with some small differences made by the Inspinia theme, but all of these is done behind the curtains so, we can just use the bootstrap classes. The positioning will again utilize the bootstrap grid system. Inside of these inputs, we can create the property called fromControlName that creates

a two-way binding to our form group values. With all the inputs done we only have to create the form-handling buttons on the bottom of the page. Where would be the return button, delete button and the save button. The return button would only have the ability to return to the previous URL with all the filters remembered with the implementation of the Angular's router. To the delete button, we would assign the disable property that would make the button disabled when the item's id does not exist (when creating a new item). Then It also should have the click event, that would trigger the function to delete the item. For that once again we can utilize the HTTP client library, just this time would call the delete request and once again provide the id inside the URL. Lastly, the save method should have the disabled property as well, only It would be disabled only if the form would not be valid. Then we would assign it the same click Event as with the delete button, this time pointing to the save method and passing it the form as a parameter. Inside the function, we would once again check whether the form is valid. If this condition passes, we would make another condition checking whether the item has some id, if it does not, we will use the post request, if it does not, we will use the put one. In both ways, we will subscribe on the result and if it passes, we would redirect the page using the router. Lastly, we would add some kind of validation before the user can save. For this, we would just add the native Angular Forms validator to the inputs we want to validate and add a property of required. Also when choosing cultureInfo for the country, we also do not want the user to add some random countries, so we can make an array of objects containing the cultureInfo and a name of a country it belongs to and insert it as the option to the HTML's select tag.

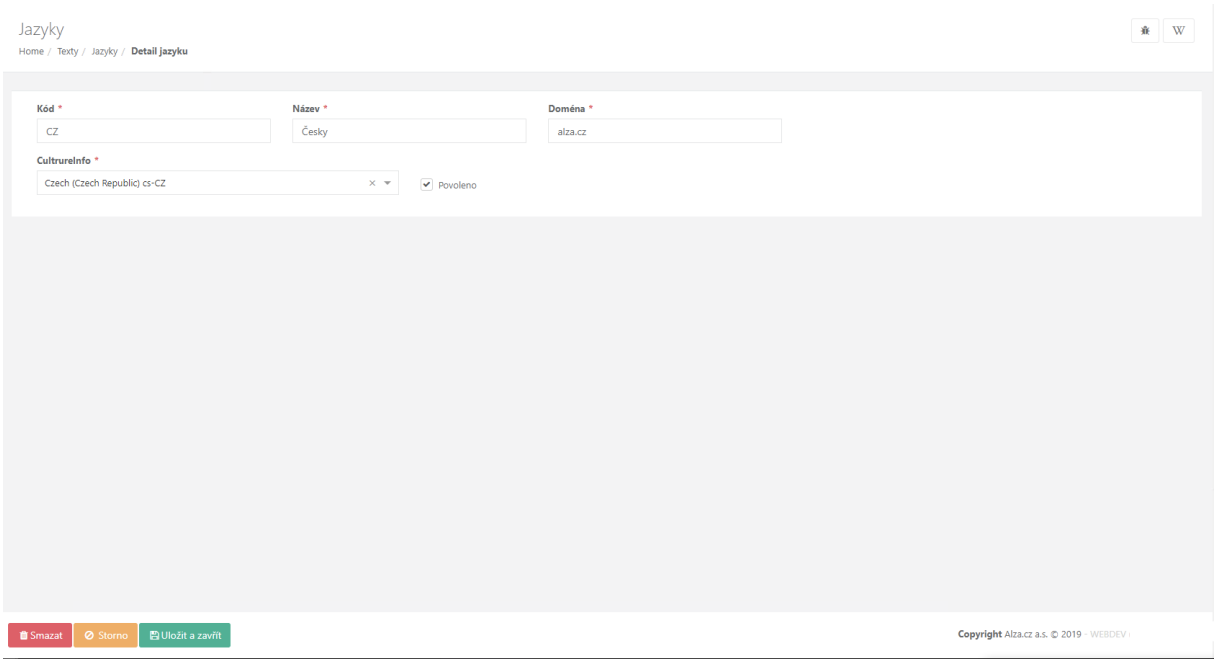


Figure 3: Language's detail view

4.12 Summary

This means now we have the collection where we have some sort of paging and filtering. By clicking on a certain row we can get to the detail of language, where we can under some validation change the values, same them to database, or delete the whole language. With this, the task is complete.

5 Book Download

5.1 Requirements

The task given was to rewrite the Book Download module from the old administration. This module serves just as a showcase of requests symbolizing downloads or info messages connected to the eBooks provided on the Alza website. The user should mainly see the date when the request was made, the code of the eBook the request is connected to, the HTTP status or the timespan of the request. The data that we are getting are generated and are synchronized on a regular basis.

5.2 Thoughts and planning

After looking at the old module, we found out, that to get the data, we need to get them from three different tables connected by foreign keys as follows. Next, we should think about the filters that should be specified and that would profit the usability of the collection if there were a lot of data. If we looked at the old module, the only specified filters were two datepickers that would specify from which date of creation we want to show the results. By default, It chose as the end the current day and as a start the first day of a month. We can follow this by for example allowing user sort data according to the timespan of the request. Also, we would allow him to choose specific HTTP statuses, that he would like to see. With this visualized we can construct the SQL procedure.

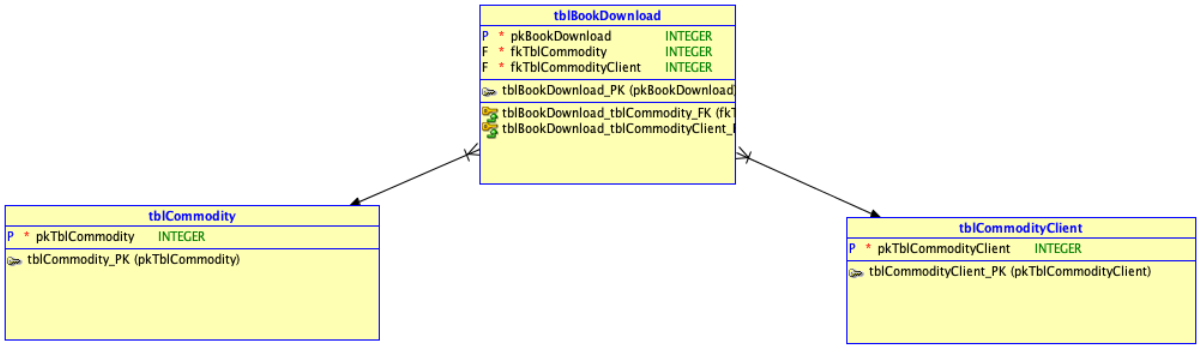


Figure 4: BookDownload table relations

5.3 Database procedures

First, we would make the create or alter command after which we will specify the name of the stored procedure. After that, we would specify all the filters that we decided were sufficient to provide the user. Next, we can use the select statement for selecting all the data needed followed by specifying the main table and a join statement that would connect the next two tables by a foreign key. Then we can add the where statement, where we would specify the filters with a

statement where we can use IIF, that would tell us whether the column should be ordered by ascendent or descendent order and specify which sort key should be assigned to it. Lastly, we want to use the SQL's count function to count the total number of items that were returned.

5.4 Model

The BL part is very similar to the one that we did with the previous task. We once again create the manager, that would call the procedure, insert the parameter properties, handle its output and add the results into the collection. The collection would be composed of return results stored as instances of our constructors. In this case, we do not even need the setter property, because in this task we are only supposed to display the data, not alter them.

5.5 Viewmodel

The viewmodel part, where our controller and service belongs is also very similar to the previous module. We only need to create the PagedCollectionResult function that would map the filters and now we can use the automapper that would handle the mapping between the business logic and the viewmodel using the data transfer object. After that, we can once again call this function from the controller specifying its API route, URL, and return the result in a JSON format.

5.6 View

The view part would this time be a little bit more difficult than last time. The structure would be the same as in the language module only without the detail page that we do not need. The collection page would have the same structure of a table as last time, but this time we would add the property of sortable column, inside of which we would specify which sort key it should refer to. This sortable column property is another global directive created for handling the sorting. It creates its visual side of a small arrow indicating that the column is sortable and it would point up or down referring whether it sorts in ascending or descending order. It changes the ordering by the click event. Also, we need to add the option to filter the data. The most valid option would be to create inputs for filtering texts, words or numbers and select tag for filtering through enums or lists. Now we need to refer these inputs with two way binding to the set of data that we got through the pager with specified HTTP request URL as in the last task. With this we can notice, that the 'from' and 'to' filter does not work properly and it still allows us to pick 'to' date that is before 'from' date and we need to specify the first day of the current month. We can fix this inside our typescript file, where we would create the function that would accept the current date with the momentjs library. Momentjs library provides a handful of nice features like isBefore function that will recognize whether one date is before another. With this and simple if statement we can provide the validation message, that something is wrong and make the form not valid. Also, we need to figure out how to defaultly set the 'from' filter to the

first day of a month. For this, we can once again utilize the momentjs library that provides us with startWith function where we can specify the keyword month and it would automatically set the value to the first day of the month.

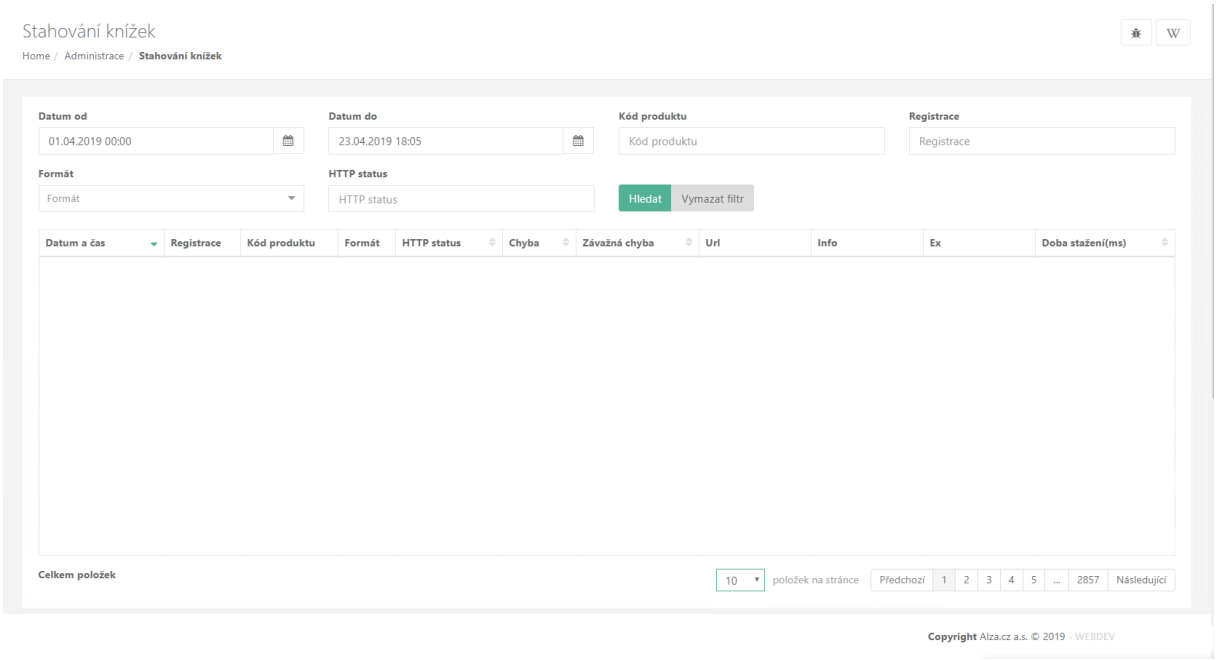


Figure 5: Book Download's collection with filters

5.7 Summary

With this, we have the module finished. It was shorter and simpler task than the last one. The only problems that we came across were to deal with three tables with simple connections, add the filtering option and to handle the date inputs inside filters.

6 Conclusion

I contributed to the company by creating modules called Language and Book Download, that are now part of Alza's information system and are used across multiple departments.

During the practice, I used the following programming languages: SQL, C#, Javascript, HTML and CSS. I already had some experience with these languages during my studies. I mostly used practical knowledge I gained from school projects like the ones assigned at Database and Information systems (DAIS), Development of Internet applications (VIA) and Development of Information systems (VIS). As a front-end programming language, I used primarily Javascript's framework Angular, that I had never worked with. At school, we only used vanilla Javascript or its JQuery library, but I had no experience with larger frameworks like Angular. I also met a team of people more experienced than me and working alongside them taught me how to work better in a team.

At the end of my practice, we have agreed that I could continue working at this company even after finishing my Bachelor's practice.

References

[1] Rick-Anderson. "Úvod Do ASP.NET Core." Microsoft.Com, Microsoft, 14 Feb. 2019. Accessed 23 Apr. 2019.
`docs.microsoft.com/cs-cz/aspnet/core/?view=aspnetcore-2.2`

[2] "Angular." Angular.Io, Google, 2010. Accessed 23 Apr. 2019.
`angular.io`

[3] Wikipedia Contributors. "Model–View–Viewmodel." Wikipedia, Wikimedia Foundation, 15 Apr. 2019. Accessed 25 Apr. 2019.
`en.wikipedia.org/wiki/Model-view-viewmodel`

[4] Otto, Mark. "Bootstrap." Getbootstrap.Com, 2019. Accessed 23 Apr. 2019.
`getbootstrap.com`

[5] "INSPINIA Dashboard." Webapplayers.Com, 2014. Accessed 24 Apr. 2019.
`webapplayers.com/inspinia_admin-v2.9.2`